VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Evolution of Digital Code

Evoluce digitálního kódu

2017

Kateřina Káňová

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Kateřina Káňová**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Evoluce digitálního kódu
Evolution of Digital Code

Jazyk vypracování: čeština

Zásady pro vypracování:

Práce je zaměřena na využití evolučních algoritmů v oblasti syntézy digitálního kódu, přesněji počítačového malware. Cílem a účelem práce je vytvořit izolované virtuální prostředí, v němž by probíhala evoluce počítačových (metamorfních) virů s použitím evolučních technik a technik symbolické regrese. Evoluční algoritmy je novotvar pro označení algoritmů, které používají Darwinovu teorii evoluce a Mendelovy teorii dědičnosti k simulaci přírodních procesů za účelem řešení složitých problémů optimalizačního charakteru. V tomto případě se tedy bude provádět syntéza virů z předem definovaných stavebních bloků. Navržené prostředí by mělo být modulární, a to s podporou připojení budoucích modulů.

Předpokládaná struktura práce je:
1. Seznámení se s problematikou.
2. Volba vhodného programovacího prostředí.
3. Volba vhodných algoritmů z oblasti evolučních technik a tvorba potřebných stavebních bloků metamorfních virů.
4. Programová realizace těchto algoritmů v jednotném GUI.
5. Vizualizace všech realizovaných algoritmů.
6. Tvorba uživatelského manuálu.

Seznam doporučené odborné literatury:

[1] Merhaut F., Zelinka I., Úvod do počítačové bezpečnosti, Fakulta aplikované informatiky, UTB ve Zlíně, Zlín, 2009
[2] Peter Szor, Počítačové viry - analýza útoku a obrana, Zoner Press
[3] Zelinka I., Oplatková Z., Šeda M., Ošmera P., Včelař F., Evolutionary techniques – principles and applications, BEN, Prague, 2008, 598 p.
[4] Zelinka Ivan , Artificial Intelligence, chapter 6 „Differential evolution", Academia, Prague, 2004, 33 p.
[5] Koza J.R.,Bennet F.H., Andre D., Keane M. 1999, Genetic Programming III, Morgan Kaufnamm pub.,ISBN 1-55860-543-6, 1999
[6] Kvasnička V., Pospíchal J., Tiňo P. 2000, Evolučné algoritmy, STU Bratislava, ISBN 85-246-2000, 2000

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. Ing. Ivan Zelinka, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

I hereby agree to the publishing of the master's thesis as per s. 26, ss. 9 of the Study and Examination Regulations for Master's Degree Programmes at VŠB – Technical University of Ostrava.

Ostrava, 1. dubna 2017 .....

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, 1. dubna 2017 .....

First I would like to thank my supervisor prof. Ing. Ivan Zelinka, Ph.D. for his patience, friendliness and open mind. I want to thank to him for his prompt e-mail replies, meetings and time spent on this thesis. I am glad that I was allowed to work on such an interesting topic under his supervision.
I am grateful to my friend Martin Velek who was so kind and helped me translate my thoughts and findings into the English language.
I wish to thank my boyfriend who gave me the strength to move on every time I got stucked.
Last but not least, I want to thank my parents who stood by me and encouraged me the whole time.

Abstrakt

Tato práce se zabývá možností spojení počítačové evoluce a samoreplikujících se struktur. Samoreplikující se struktura je podobná počítačovému viru, avšak v několika prvcích chování se podstatně liší. V úvodu práce je seznámení s problematikou počítačových virů a evolučních technik, zejména se zaměřením na algoritmus SOMA. Ve střední části je popsána implementace problému. V závěru se pak nacházejí proběhlé testy a komentování jednotlivých výsledků.

Klíčová slova: syntéza digitálního kódu, virus, evoluce, SOMA, diplomová práce

Abstract

This master thesis focuses on the possibility of connecting computer evolution and self-replicating structures. The structures are similar to computer viruses but they differ substantially in severeal aspects of behaviour. At the beginning of the thesis the familiarization with the issue of computer viruses and evolution techniques takes place. Special attention is paid to the SOMA algorithm. The middle part describes the implementation of the problem while the final section of the thesis presents and comments the results of the performed tests.

Key Words: synthesis of digital code, virus, evolution, SOMA, master thesis

Contents

List of symbols and abbreviations

FIFO	–	First In First Out
SOMA	–	Self-Organizing Migrating Algorithm
SS	–	Self-replicating Structure

List of Figures

List of Tables

Listings

1 Introduction

Due to still emerging technologies and due to modernization and digitalization of almost all processes, more and more information about us and about our lives are stored in digital form. Talking about numbers of our credit cards, medical records or information about our current position, all of these data are saved on servers and therefore there is a possibility of a hacking attack and stealing data.

Computer systems are getting more and more sophisticated so viruses, which attack these systems, have to get more sophisticated too. On the other hand, some of the systems are insufficiently secured despite warnings of public about privacy and about posting their data. These systems are an easy target of an attack.

The fact, that a hacker finds a flaw in a system, that can be abused, doesn't have to be a bad news. There are "white hackers" as well, hackers who help securing systems. They report flaws they find so they can be fixed. Unfortunately, some companies don't care about these warnings and don't fix the flaws in their systems.

To protect your computer, you can obviously use some antivirus protection. There is a great number of companies that specialize on the issue. To protect your computer and to destroy a virus, an antivirus program have to find the virus first. This is probably the greatest challenge, because there are many modifications of each individual virus. There are also viruses that can change their body by themselves.

The aforementioned issues are the main topic of the thesis. Is it possible for a code structure to evolve, to change itself to be better than its older variant? Computer viruses are one of the best structures to test this idea. Hence the self-replication structure (SS) is defined as a virus. **SS isn't a real virus – it does not pose any threat, it's only there to test files that are created only for testing purposes. Users can set testing files and HDD partitions for the attack and so this "virus" is strictly controlled. At the end of the program execution, all infected files are removed from all partitions.**

The thesis is divided into three main parts. The first one aims to provide theoretical information that serves as a basis for this work. In the second part you can read about implementation of the program. The results of the work are summarized in the last chapter.

2 Theoretical part

This section is a brief overview of the basics needed to understand the practical part. It is divided into two smaller thematic subsections.

The first subsection focuses on viruses, their types and methods. It specifies which of these methods were used, but their actual implementations are shown in the practical part of the thesis.

The second subsection explains evolution and the evolution methods. The methods that were implemented are described in detail. The principle of these methods is explained there, but again the actual implementations are shown in the pratical part.

2.1 Viruses

Although the word "virus" is widely used, it is better to use the term "malware". If we talk about malware, we mean harmful code that works to an attacker's advantage. Its specific functions allow us to classify it in many ways. One of the classification groups are viruses.

Most of this part is devoted to viruses because a virus (in a very simplified way) is used as an entity for the evolution process in the thesis.

2.1.1 Similarity to a biological virus

The idea of a computer virus is very similar to a biological one. [1] [3] It is a cell of information that is able to find a host and to spread. [2] Although there are some similarities between a computer virus and a biological one, there are some distinctions too. The following Table 1 is an overview of the diverging attributes.

From the technical point of view we can define a virus as follows:

A virus can be described by a sequence of symbols which is able, when interpreted in a suitable environment (a machine), to modify other sequences of symbols in that environment by including a possibly evolved copy of itself.

That means, that viruses don't spreed among computers over a network (as worms do) but they infect files in PCs. So how can viruses be relocated from one PC to another? In fact there are many ways a virus can find your PC. [3] Some of them are given below:

- Sharing music, files or photos with other users.
- Visiting an infected website.
- Opening a spam e-mail or its attachment.
- Downloading free games, toolbars, media players and other system utilities.
- Installing mainstream applications without fully understanding their license agreements.

Table 1: Divergence of computer and biological viruses

Biological virus	Computer virus
They can't self-replicate. They need infected cells.	To reproduce they need infected files.
They attack specific cells.	They attack specific files.
They change the infected cell's DNA to replicate.	They change the infected file's data to replicate.
They take control of part or of the whole cell.	They are executed before the original file.
The cell is usually not infected twice by the same virus.	Most viruses don't infect the same file twice.
Symptoms may not be exhibited or could be delayed.	Symptoms may not be exhibited or could be delayed.
Viruses mutate. It is more difficult to find them and cure them.	They can mutate or include "safeguards". Finding and destroying them is more difficult.
Cells can be vaccinated.	Files can be protected.

Although there are many antivirus programs on the market, they only deal with consequences of users' behaviour. Thus, if you are careful, you make work of your antivirus much easier.

2.1.2 Virus types

Virus itself is a harmful piece of code often with a destructive payload. Even if it is not destructive, it is still harmful because it steals computing time and resources to do things you don't want your PC to do. There are many virus types depending on the way they spread, the objects they infect, the way they infect these objects etc. [6]

Differences in payload

By a virus payload we mean its special behaviour. Taken from [4], page 46, it states:

'The existence of a payload – in other words an offensive procedure – is not an essential feature in characterizing a virus.'

Some consider a virus behaviour (e.g. spreading) to be a part of its payload. Common use of the word 'payload' means an additional function with a special purpose – it may send data to a hacker, watch a user's keyboard etc. The following section treats payload as a special functional part of a virus body.

Payload execution usually takes place by the end of virus lifecycle, after it has spread. Sometimes it is executed near the beginning, but there are some disadvantages to this approach. If such a virus gets discovered by e.g. an antivirus while performing its payload, it is usually terminated and thus has no chance to spread. That is why it is better to let viruses spread first

and do some special activity later on. [4] Payload can be delayed and may wait for a special condition – for example a specific date or a number of successful infections. [4]

We can simply divide virus payloads into lethal and non-lethal ones. [4] The lethal ones are those that corrupt files, steal data, corrupt or destroy systems or violate data integrity etc. Non-lethal ones are e.g. displaying some pictures or drawing a user's attention to such-and-such topic.

Targets

Viruses usually infect executable files (.EXE, .COM) and they abuse specific structures of these files. On the other hand, there are boot viruses that attack boot sectors or macros in documents created with Microsoft Office tools. Throughout the thesis only viruses attacking .EXE files are studied – for the purpose of creating SSs. Main focus is put on principles that are used by viruses attacking .EXE files. Other virus types are therefore not subject of further analysis.

Ways of infection

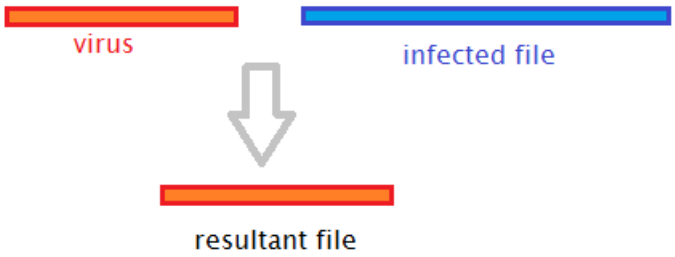
In general, viruses spread from file to file, that means they don't use network as worms do. They spread when the infected file with their code inside gets executed. There are, however, many ways in which files can be infected.

- **Rewriting**

  The rewriting virus is the simplest virus in terms of infection. It does not allow restoring of the infected file. The infected file is simply rewritten by the body of the virus while the body of the original program is lost. The figure 1 shows the rewriting principle.

  Using this method the resultant file can be either larger or smaller than the original one.

Figure 1: Rewriting method of infection

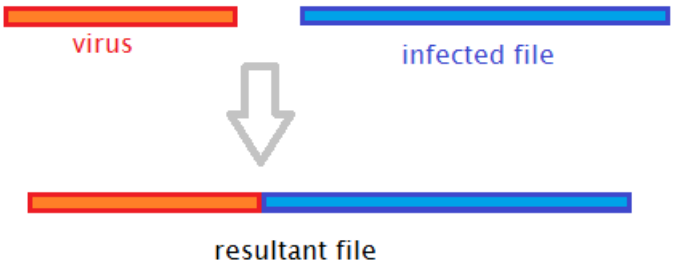


- **Prepending** This method prepends virus code to the beginning of the infected program (so it gets executed before the original code) as you can see in the figure 2. The original program is usually run from the virus body immediately after the virus is executed. Should it be run later, the user could suspect something and might start looking for the cause of the delay or he/she might scan the device with an antivirus.

  Using this method the resultant file is always larger than the infected one.

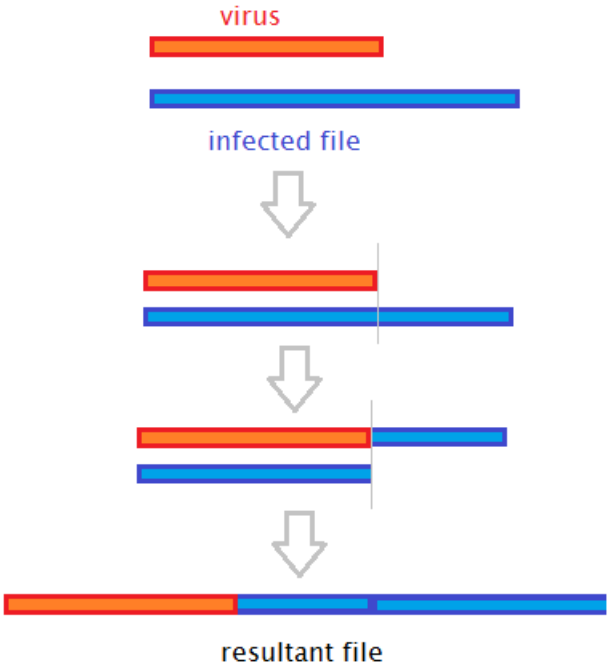
The advantage is that the virus does not delete the content of the host file and it is fully restored so the user has no idea of the pending virus infection.

Figure 2: Prepending method of infection



- **Parasitical** The principle here is very similar to the one employed in the prepending method. The difference is that the original code isn't moved to the end, but split at the end of the virus body. The first part of the infected file is put to the end of the file. When the virus is run, it connects these parts together again and executes it in the way a prepending virus does. The figure 3 explains how it works.

Figure 3: Parasitical method of infection



2.2 Other malware types

Although virus is a cornerstone of SS, some other well-known types of malware are given here as well.

2.2.1 Trojan horse

Trojan horse is malware similar to a virus with a single difference – it cannot replicate. In a simple way we can say that trojan horse is the payload of a virus (Description of payload 2.1.2). Trojan horses are mostly executable files – so they do not need a host file. To execute trojan horse code, the file containing a trojan horse only needs to be executed (by a user or by a process running on the device). This is the reason why trojan horses try to look like useful or interesting applications – to make users execute them. Hence the name.

Although a trojan horse cannot spread by itself, it is very often a part of a virus. This technique alows trojans to spread so you can encounter a trojan horse more often than you would expect. [1]

Basic types of trojan horses are summarized below.

- **Destructive trojan** is a type of trojan that has a destructive impact – e.g. deleting files or formatting drives.
- **Keyloggers** are programs that monitor pressed keys. They collect them and send them to the attacker. This type of infiltration can also be classified as spyware.
- **Dropper** is like a giftbox hiding trojans in it. Droppers usually carry .EXE files. When executed, they let harmful programs spread in the device.
- **Downloader** is a type of trojan horse similar to dropper, but it does not carry all of the harmful code with it. When downloader is executed, it simply downloads the needed content. Downloaders use predefined URLs to download content from. Great advantage is that if the content on the server is changed, the same downloader downloads the updated code and starts behaving differently.

2.2.2 Backdoor

Backdoor is a client-server application, whose abilities are very similar to commercial products such as pcAnyWhere, VNC or Remote Administrator. [1]

The uniqueness of this type of malware is that it behaves in a way that a user cannot notice them. Controlling a device remotely may not mean anything bad, but if the activity is harmful, then we call the manipulating person a 'remote attacker'. [1]

How does it work?. The client side of the application is controlled by an attacker, while the server side is in a victim's device. Communication usually uses TCP/IP, which means that the attacker can be far away from the server side of the application. [1]

2.2.3 Worm

Worms are structures that work in a different way than viruses do. They don't infect files, rather they infect network packets themselves. That is why worms are not recognized by common antivirus programs. One of the most common impact of a worm infection is network congestion.

2.2.4 Spyware

Spyware is a type of malware that serves attackers as a spy. It collects information and send it back as in the case of backdoors. Unlike backdoors, spyware usually sends static data such as visited web pages or installed programs. The main reason why these programs exist is supposedly collecting data for targeted advertising.

2.2.5 Adware

Adware is something that makes you annoyed while working with your device. Adware should act as an advertisement on the web. Mostly the user agrees to installation of adware, but it can be a part of some product as well.

2.2.6 Phishing

Phishing attackers use web pages and forms and they persuade users to fill in their personal information. They achieve this by threatening users to make them do that. For example an attacker sends a message to a user that their account will be deleted or banned if they don't fill in the form. The problem is that these web pages and forms are very authentic so users have no idea they are being attacked.

2.2.7 Where to find information on malware

There are plenty of servers dealing with various topics on malware. Almost every antivirus company has its own website, own virus database and they present basic information about them. You can test files you have in your device at https://www.virustotal.com. Web pages symantec.com show some information too.

Interesting summary is presented at http://www.avgthreatlabs.com/ww-en/virus-and-malware-information/ where you can find a map showing areas of virus detection. On the main page there are up to date information about the situation in the world for the past week. The figure 4 shows information from the first week of April 2017.

The ESET alows you to scan you PC online from https://www.eset.com/int/home/online-scanner/.

The company Symantec shows the number of malware which appeared depending on the malware type. For example email threads from last year [5] are shown in the Fig. 5.

Figure 4: Statistics taken from avgthreatlabs.com

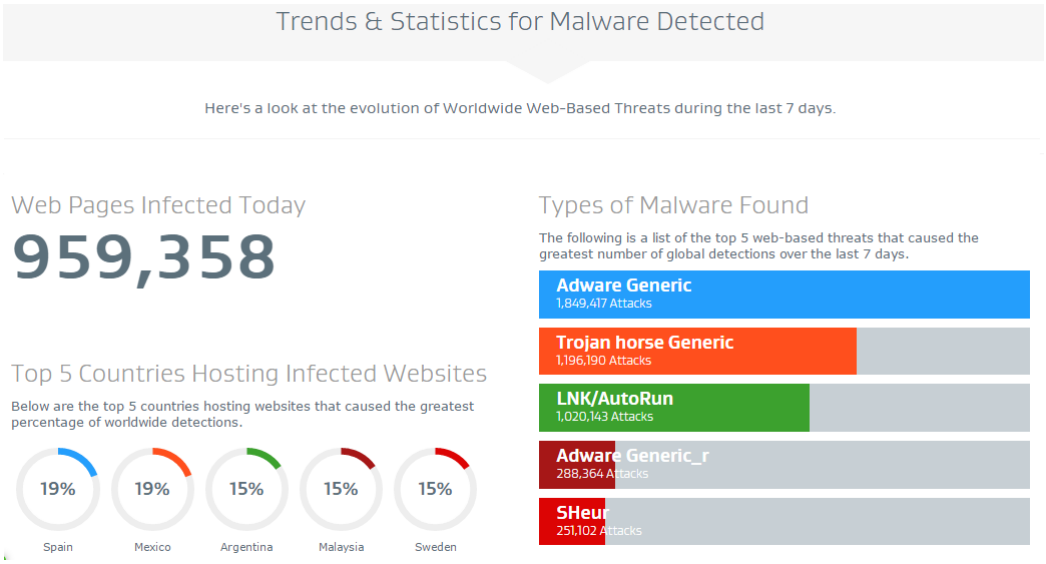
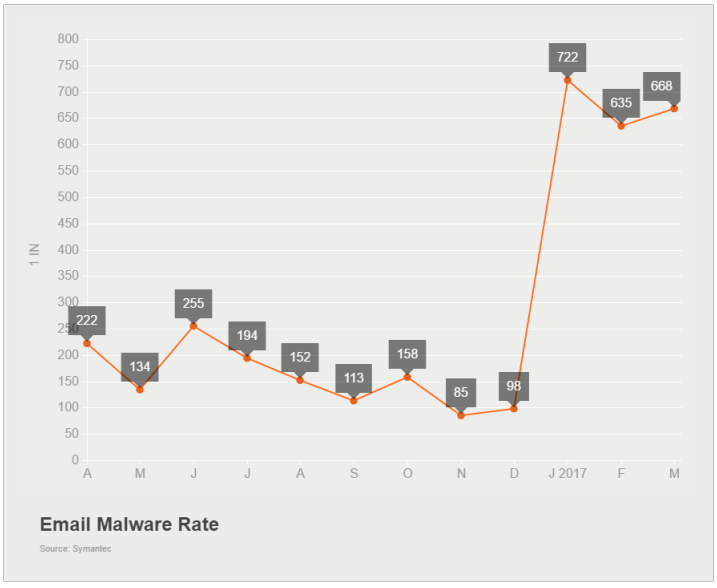


Figure 5: Statistics taken from symantec.com



2.3 Evolution

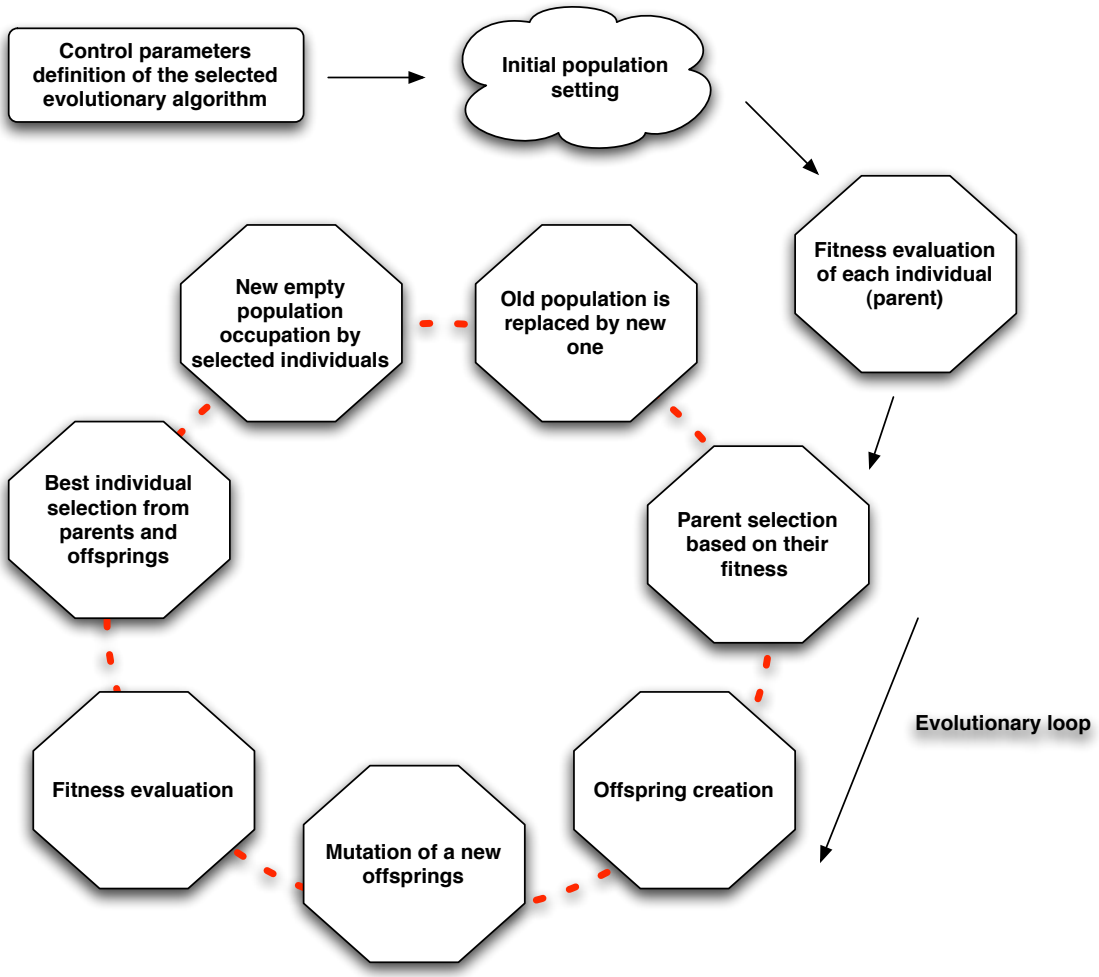
Evolution process is a term well-known from biology the definition of which sounds as follows: *'Evolution is the process by which the physical characteristics of types of creatures change over time, new types of creatures develop, and others disappear.'* [8]

In the digital world, the first signs of evolution techniques can be dated to the early 1970s. At this time, genetic algorithms were discovered. A few years later the 'evolution strategies' were successfully used. [9]

Evolution is a simulation process that moves forward to find new (and better) solutions. The evolution of a digital structure is analogous to breeding a new biological species, e.g. a new dog breed. Entity is an instance, i.e. a particular dog (or digital structure). The best entities are crossbred to spread their genes. The goal is to get a new entity (offspring of the parents) that should be better than its parents and closer to the desired features. The best offsprings become parents and the whole process repeats until we get a perfect entity. Unfortunately we don't usually get a *perfect* entity in the real world. So we have to settle for the best solution after e.g. a limited number of evolution cycles or after a certain amount of time.

The generalized view of evolutionary algorithm is shown in Fig. 6.

Figure 6: Evolutionary algorithm



Evolution in the digital world is very similar. The purpose of evolution is to get the best solution to a given problem. In the thesis evolution is applied in order to synthesize SSs with the best possible attributes.

To compare two entities and to find out which one is better we need to establish a rating. This rating is called a **fitness** value and it is a result of **cost function**. [7] For example to evaluate which dog is better we should consider its height, color, behaviour, hair length and so on. If we want to breed a new virus, such a rating would make no sense. That means the cost funtion (rating system) depends on the problem we are trying to solve. If we talk about a virus, we want to have a rating that depends on the length of code, the execution time etc.

As a general rule we can assume that the higher the fitness value the better the entity. The fitness value is normalized to be in the interval from zero to one.

In the thesis we want to obtain the best SS, which is composed of several parts. We seek an entity with the shortest code, the shortest time of execution and the smallest number of errors. And so in this case the lower the cost function value the better the entity.

2.4 Principle of the SOMA algorithm

To explain how SOMA works it is necessaary to know basic terms from the evolution theory. A brief overview follows to refresh them.

Entity – an entity used in the evolution process (we want to evolve SSs, so the entity here is one particular SS).
Generation -– a group of entities created in a single evolution cycle.
Fitness value -– the value that says us how "good" a particular entity is.
Migration loop – the process of moving entities and finding a new generation.

The SOMA algorithm caught my attention and so I chose it for the purpose of the thesis. In a nutshell we can say that it doesn't work on the principle of classical crossbreeding of parents, but it resembles some kind of a migration. Entities do communicate with each other to determine which one of them is the best of the generation. The reason why it is so close to migration algorithms is that many offsprings are created from two parents. That means we get a new group of offsprings from the two parents – but only the best one will be a part of the next generation. Thus it seems as if the original parent move – migrate – somewhere. Before we delve into details, it is necessary to understand basic aspects of SOMA. These terms and their descriptions are shown in Table 2.

PRTVector has the same purpose in SOMA as mutation has in evolution algorithms. The number of PRTVector elements is the same as the number of attributes defining an offspring. For each of these attributes a random number is generated in the interval from zero to one. This generated number is compared with the PRT parameter. If it is less than the generated number, zero is written to the PRTVector, one otherwise. These binary values give us information about mutation – if zero is written at a position in the PRTVector, the parameter of the offspring at the same position will NOT be mutated.

Table 2: Parameters of the SOMA algorithm

Term	Explanation
Leader	The best entity in a single generation.
PathLength	The length of the path, the distance between an entity and the leader.
Step	The number of steps needed to get an entity to the leader.
PRT	The "mutation" parameter, here called perturbation.
D	The number of parameters taken by the cost function.
PopSize	The size of the population.
Migration	The count of migrations, terminating parameter.
MinDiv	The difference between two successive best solutions, terminating parameter.
PRTVector	The perturbation vector.

Let us go over the example below to see how the creation of a PRTVector looks like. We work with 3D space so the attributes of an entity are coordinates x, y and z. We set the value of the PRT parameter to 0.2.

- First, we have to generate some random numbers.
  We generated rand1 = 0.523, rand2 = 0.127 and rand3 = 0.256.
- Now we have to compare those values with the PRT value.
  PRT $<$ rand1 ... 0
  PRT $>$ rand2 ... 1
  PRT $<$ rand3 ... 0
- We obtained some values. Now we create a new PRTVector using those values.
  PRTVector = (0, 1, 0)

Now we have a PRTVector. This vector is relevant only to a single entity in a single migration cycle. It means that every time a new offspring is generated, a new vector is needed in the process.

AllToOne as an essential method as it forms the basis for both AllToAll and AllToAll Adaptive methods, that is why it is described first. AllToOne method was also selected for the implementation. SOMA has caught attention of many people and numerous modifications are available today. Some of the well-known modified versions are AllToOne, AllToAll and AllToAll Adaptive methods. These are described further in this chapter.

2.4.1 SOMA AllToOne method

This is one of the simpler methods and as such it is suitable for explanation of the SOMA principle.

To have a better idea of the process, it is divided into several steps:

1. Consider 3D space. Generate some entites within. These entities form the first generation.
2. Find the leader – evaluate the cost function for each entity. An entity with the best fitness value becomes the leader.
3. Migrate all existing entities to the leader's position. (Theleader does not migrate.)
   (a) Pick an entity.
   (b) Have parameters: t = 0, bestPosition = starting position and bestFitness = migrating entity's fitness.
   (c) While t < PathLength
      i. Generate PRTVector
      ii. For each attribute do:
         - newValue = entityAttributeValue + (leadersAttributeValue - - entityAttributeValue) * step * t * PRTVector[position of the current attribute]
      iii. Get fitness of the new entity
      iv. If the new fitness is better than bestFitness
         - newBestPosition = current new position
         - newbestFitness = fitness of the new entity
      v. t = t + step;
   (d) If there is an unmigrated entity – take it and continue with step 3.
4. When all the entities have migrated, move them to the best new position found.
5. Get a new leader of the generation. (The leader of the previous generation is a part of the new generation too).
6. If the terminating condition is not met, continue with step 3.
7. Program has ended. The leader of the last generation is the best found entity.

The above resembles swarm intelligence algorithms. Entities are capable of low-level communication (they are able to find the best one among themselves – the leader) and so they can calculate the solution, i.e. the final leader, as well. The information exchange is an advantage because the algorithm converges faster to a sought extreme. This approach of course has some disadvantages. The algorithm may get stuck at a local extreme, but this nuisance can be overcome if the path and the step parameters are set correctly. When a new leader is found, all entities migrate to the current leader's position and thus the rest of the space is left unexplored. This problem can also be eliminated, that is by placing the first generation of entities to different parts of the space.

2.4.2 SOMA AllToAll method

Another way to a solution may be the usage of the AllToAll method the principle of which is much the same as the AllToOne method. The difference is that no leader is chosen. Each entity migrates to all other entities in the space. After all migrations have taken place every new potential position (an offspring) are considered, the entity is then moved to the best one.

Thanks to this kind of migration, a larger area is searched. On the other hand, if the given function has simple and not wavy progression, it is likely that the AllToOne variant would yield the same result in less time. To choose the right method it is important to know the problem you are dealing with.

2.4.3 SOMA AllToAll Adaptive method

The AllToAll Adaptive variant draws on AllToAll as its name suggests. The difference is that if a better position is found, the entity moves immediately. The following calculations can thus operate on the new position at once. This way the time needed to find the best position is shortened. To the contrary, if an entity is headed directly to an extreme, it will change its position every time it moves, which costs time.

3 Practical part

The goal of the thesis is to program a self-replicating structure (SS) that can be evolved by the SOMA algorithm.

The whole program is written in the C/C++ language. The program comprises two parts – the evolutionary part and the part that represents SS and its functions. The following text describes the structure and the functionality of both these parts.

3.1 Self-replicating structure

We define a self-replicating structure as a virus because of their similarities. Before we dive into the nuts and bolts let me stress that the SS used in the thesis is only **a very simplified virus that does NOT contain any potentionally dangerous payload. Its behaviour is strictly controlled.** This 'pseudovirus' is only used for the purposes of testing digital code evolution. The whole program, in fact, is used for scientific purposes only.

3.2 Requirements and instructions

Code requirements The source code is well-structured and easy to extend. These requirements apply mainly to the SS part because there are many methods with various functionality. These methods are combined into a single working object. To work with the results it is vital to clear up the structure of the obtained SS, the parts and methods used etc.

The project consists of two logical components that communicate with each other – the evolution component generates a string representing the body of a SS. The string is sent to the fitness handler component that does the computing and gives back the result to the evolution component. Then, based on the returned values fitness is determined (you can read more on fitness in chapter 2.3 on page 20) and the offspring is evaluated.

How to run the program

To run the program several attributes have to be set. You need to customize the file SOMAconfig.csv the structure of which is shown in Fig. 7.

As you can see there are many attributes. Table 3 clarifies what each attribute means.

The file has to be in the same folder the program is executed from.

Figure 7: Example of SOMAconfig.csv

	A	B
1	Offsprings count:	3
2	Path to tester file:	D:\virusTest\tested\KTHelloWorld.exe
3	Number of copies of tester file:	3
4	Drives to infect (write only letters separated with space):	C D
5	Step:	0.36
6	Prt value:	0.5
7	Path length:	4
8	Maximum number of payloads:	4
9	Maximum count of migration loops:	5
10		

Table 3: SOMA's configuration file structure

Attribute	**Description**
Offsprings count	The number of offsprings generated for each generation.
Path to tester file	The file that should be spread across drives and used for testing the SS.
Number of copies of tester file	The number of copies the tester file should be copied and spread across drives.
Drives to infect (letters separated with a space)	The drives to spread the tester file across.
Step	The number of steps needed to get entity to the leader.
Prt value	The "mutation" parameter – here called perturbation.
Path length	The length of the path, the distance between an entity and the leader.
Maximum number of payloads	The upper bound of payloads a SS can have.
Maximum count of migration loops	The upper bound of the performed migrations.

3.3 Evolution part

The evolution component of the program takes care of evolution of the SS body first generating offsprings and calculating their fitness values. The source code is broken down into logical components, i.e. classes FitnessHandler, TestHandler and Evolution. Each of them are outlined in the following sections.

3.3.1 FitnessHandler class

FitnessHandler computes the fitness vaule of an offspring. To understand the way it works it is crucial to realize which attributes are required to calculate fitness. The parameters and their meanings are provided in Table 4.

Because an entity with the shortest execution time, the smallest number of missed files, the shortest length, the lowest penalty and the lowest number of failures is desired, the best offspring is the one with the lowest fitness.

Table 4: Fitness value explanation

Attribute	Description
Time	The time needed to run the whole entity.
Files missed	The number of files the entity is unable to find (penality).
Length of SS	The length of the entity, which is a sum of characters the entity is comprised of.
Method penalty	The penalty incurred by the entity's using a method marked as penalized.
Infection failure	The penalty incurred by the entity's inability to infect a file or by infecting a file twice.

Time

Time is represented as a number of miliseconds. It measures the whole SS lifecycle with all the methods it includes. The lifecycle of a SS is defined in chapter 3.4.11 on page 39.

Because finding all the files in a device takes a substantial amount of time, this method is not executed each time a SS is tested. All file-searching methods are tested on the program start up, their duration measured and saved for future reference – computing the corresponding fitness value.

Files missing

The number of files that should be infected are specified in SOMAconfig.csv. If an offspring is unable to find all of them, the count of missed files is used to penalize the offspring.

Length

The length of a SS plays a role in the fitness calculation too. If a virus is too large, it is much easier to find and it may take a lot of HDD space. The size of a SS offspring is not defined in bytes, but rather in characters the comprise the offspring body.

Method penalisation

There are also fake methods that do nothing or they include the behaviour of a SS. For example there is a method that causes inability to spread. This kind of misbehaviour is not wanted so if any such method is used, the corresponding offspring gets penalized. Evolution should take care of those offsprings and rule them out when performing selection for a new generation.

Infection failure

There are two types of an infection failure. One is that the file cannot be infected and the other that the file was infected twice. Both are unwanted – if a virus cannot infect some files, it probably doesn't spread so well or perhaps slower. As for the case when it infects a file twice, it causes recursion and as a result adds one more infection every time the virus is executed. Consequently the file gets bigger and bigger and the execution of the original content is slowed down.

The failures are added up and the resulting number is used to penalize the SS.

Calculating fitness

For each entity the variables described above are summed (including penalties) to form the final fitness value. The lower the fitness value the better the chance for an offspring to be the best of its generation.

3.3.2 TestHandler class

Because testing is a complex process, the class TestHandler is there to facilitate it. The biggest issue is handling tester files. Tester files have to be put in place. The first SS can then be tested, that means the files are infected. Before testing any consecutive SS, the files have to be restored and other files created during the test have to be removed. That is the main task of TestHandler.

Although TestHandler is used primarily for tester files handling, it also handles testing of the search methods.

Spreading files

On the program start up TestHandler tries to find directories in the device to put the tester files in place. It works in a simple way: first it locates all directories in the device and saves them to file allFolders.txt – one folder path per line; then it continues with generating random numbers. Those numbers serve as accessors (line numbers) to allFolders.txt. If a selected path is inaccessible e.g. due to access restriction, another number is generated. The process repeats until the right number of suitable paths is reached.

The paths are saved to file controlFiles.txt. Tester files are then copied to the specified locations making it ready for evolution.

Reseting files

When the program is done testing an offspring, the tester files are most likely infected and they need to be reset for another offspring waiting to be tested. That means all infected tester files and the files that were created for the sake of the test are removed. Then the tester files are copied to those locations again so another offspring can be tested.

Tester files are not in any way affected by the testing. Even if an original tester file is located on a drive that is attacked, it endures. This way users do not need to worry about the location of their original tester files – whether they are on an infected drive or not.

Deleting files At the end of evolution tester files are no longer needed. All of them are, therefore, deleted (including test supporting files).

Just like when performing the file reset the original file is left intact and on the same location as originally defined.

Search methods' execution time premeasurement The TestHandler class measures file-searching methods' execution time. If the search took place every time an offspring is tested, it would take considerable amount of time. That is why the execution time is stored for future use to speed up the process of testing.

Every method is executed and measured three times with the average being taken. The averages are stored in class ConstantsHandler. The number of found files as well as the number of missed ones is stored there for each of the methods too. Those two values are then used as constants throughout the rest of the program. No file-searching method is run past this point.

3.3.3 Evolution class

Evolution class is the main class of the project. When executed setting procedures run first, then evolution itself and at the very end cleaning methods are called.

Summary

While evolution is running data about the process are logged to a summary file (summary.txt). At the end the summary file is loaded and presented to the user. As you will see in Fig. 8 the summary file carries a lot of information.

Figure 8: Example of summary.txt

```
---------- EVOLUTION OF SELFREPLICATING STRUCTURES USING SOMA ------------

Generated offsprings:
id: 0
    body:
    SeparateExe layer: 2. method for rewriting virus
    SearchingForDrives layer: 1. method using GetLogicalDrives() method
    SearchingForFiles layer: 2. method using recoursive approach with HANDLE, files are writen to file at
        the end - when all files are found
    FilesToInfectHandler layer: 2. method using parallel approach to files for infection
    InfectionChecker layer: 2. method checking length - if same, returns that it is infected
    InfectExe layer: 2. method for rewriting virus
    Payload layer: 6. method count fibonaci for 100 iterations
    Payload layer: 5. method writes line of text into the file
    Payload layer: 5. method writes line of text into the file
   fittness: 31232
id: 1
    body:
    SeparateExe layer: 4. method is fake method that does nothing and is penalised
    SearchingForDrives layer: 2. method using system command
    SearchingForFiles layer: 1. method using system command
    FilesToInfectHandler layer: 2. method using parallel approach to files for infection
    InfectionChecker layer: 1. method checking length, checks if other exe file is appended (good for prepended virus)
    InfectExe layer: 4. method is fake method that does nothing and is penalised
    Payload layer: 5. method writes line of text into the file
    Payload layer: 2. method does nothing
    Payload layer: 2. method does nothing
   fittness: 18806
id: 2
    body:
    SeparateExe layer: 1. method for prepended virus
    SearchingForDrives layer: 1. method using GetLogicalDrives() method
    SearchingForFiles layer: 1. method using system command
    FilesToInfectHandler layer: 2. method using parallel approach to files for infection
    InfectionChecker layer: 3. method checking length - if same or greather, returns that it is infected
    InfectExe layer: 1. method for prepended virus
   fittness: 17697
```

The 'id' uniquely identifies an entity. The 'body' section of the file adheres the structure **layer: method description**. Layer captures the main functionality of the used method while method description provides detailed information on the method. Outside the 'body' section there is also fitness, which is calculated depending on the entity's body. After every migration loop the best entity is recorded in the file as well as shown in Fig. 9. Distance constitutes the difference between the previous and the new leader's fitness.

Figure 9: Example of a generation summary

```
Leader id: 5
Leader cost function: 16219
distance: 85
```

3.4 SS part

Originally there were two ideas of evolution of SSs that came into question. The first one was evolution on the bit level, the second one worked with methods as wholes.

As to the first approach, due to mutations a lot of nonfunctional code may appear. Such code would either be executed with failures[1] or not at all. That could render fitness checking much more difficult because recognizing which behaviour of the mutated entity is right and which is not is, by its nature, a challenging task. Thus the final solution differs from those concepts.

The body of a SS is comprised of blocks of code structured in layers. Each layer has a number of methods that can be used to create the whole body of a SS. It is like putting building blocks together to create one functional entity. This way no nonfunctional code can be created, which makes testing much easier. Some layers communicate with other layers. Such layers have input and output parameters in place for this purpose. Not all layers need to communicate, however. Those simply do not have the parameters.

3.4.1 Representation of a SS body

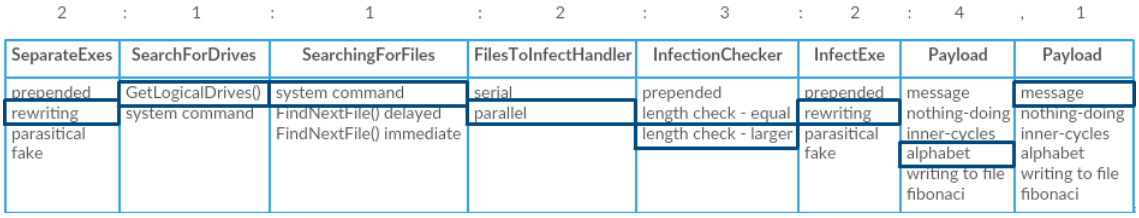
A SS body is a string of numbers separated by colons. Every number **position** determines a layer and the **value** of the number selects the method from the corresponding layer. For example, consider the virus body '5:2:4'. It conveys the information that the fifth method of the first layer alongside with the second method of the second layer and the fourth method of the third layer should be used. Each numeric combination makes a unique virus body with a specific behaviour.

Payload layer is an exception. It is the only layer from which none, one or more than one method can be used. That, of course, means that the method count can differ. Payload methods are separated with a comma. For example the payload layer '1,4,1' tells us to use the first method, the fourth method and then the first method again.

To have a better idea the body may look like this: '2:1:1:2:3:2:4,1'. The given example of a SS is used in Fig. 10 to illustrate how the methods indexes work. The string representation is only used from the source code. The user gets the SS bodies in a human-readable form as shown in Fig. 8.

[1] It could possibly introduce various other kinds of misbehaviour.

Figure 10: The body of example SS



3.4.2 ConstantsHandler class

This class persists the data collected throughout the whole application. Table 5 shows the records it creates together with their meanings.

Table 5: ConstantsHandler records

Attribute	**Description**
Layer count	The number of layers used in the definition of a SS body.
Executin path	The path from which the program was executed.
Execution folder	The folder from which the program was executed.
Size	The size of the executed program.
Method counts	The number of methods in each layer.
Tester file	The path to a file that is spread across the device to test SS offsprings.
Number of copies	The number of spread out tester files.
Disks	The disks marked for infection.
Control file	The name of the control file. Set by default to 'controlFile.txt'.
Listed file	The file with the found tester files.
Payload count	The maximum number of payloads used in a single SS. This value is set by the user.
Non-infected files	The number of files that were not infected by a SS.
Times and files	A file-searching method's execution time 3.3.1 and the files it was able to find.

3.4.3 Virus class

Virus class embodies the SS and contains methods essential for the SS to run. The class maintains the string representation of the SS body. There are methods designed to determine indexes to each layer. The methods that are actually used in the program are accessed with the retrieved indexes.

This class executes the virus body, which consists of methods from specific layers. The following part describes those layers and methods. Listing 1 shows the structure of the SS body control method.

```
int Virus::runVirus(int payloadCount){
  milliseconds start = duration_cast<milliseconds>(
    system_clock::now().time_since_epoch()
  );
  searchingForDrives->perform(getIndex(2));
  handler->perform(getIndex(4), getIndex(5), getIndex(6));
  int notInfected = ConstantsHandler::getNotInfectedFiles();
  separateExes->perform(getIndex(1));
  if (payloadCount != 0){
    payload->perform(getIndexes(7));
  }

  handler->perform(getIndex(4), getIndex(5), getIndex(6));
  int twiceInfected = ConstantsHandler::getNumberOfCopies() - notInfected -
     ConstantsHandler::getNotInfectedFiles();
  ConstantsHandler::setInfectionFailures(notInfected + twiceInfected);

  milliseconds end = duration_cast< milliseconds >(
    system_clock::now().time_since_epoch()
  );

  long long int lasting = end.count() - start.count();
  lasting = lasting + ConstantsHandler::getTimesTaken(getIndex(3));
  return lasting;
}
```

Listing 1: Method running SS

3.4.4 SeparateExe layer

Inputs	—
Outputs	nothing or a new file with the orignal program

This is the first layer in the body string. It separates the virus body from the original code (if possible). The methods from this layer have to correspond to the methods defined in InfectExe layer – one undoes the other and thus their indexes have to be the same.

Some of the layer's methods are there to recover the original code from infected creating new files in the process. The name of the files are composed of the tester-file name and the recovery time in miliseconds. This guarantees that any two generated files in the same folder will have different names.

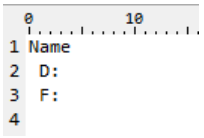
The original file is executable, but it is not run during the testing because it would slow down the application and would yield no data that may be relevant to fitness calculation.

3.4.5 SearchingForDrives layer

Inputs —
Outputs file disk.txt

This layer searches all of the drives connected to the computer. The output is the file named 'disk.txt'. The drives found in the device are written into that file. An example of the file is shown in Fig. 11.

Figure 11: Example of file disk.txt



The layer consists of two methods. These method differ in their approach to getting the information on the connected drives.

Use of Dirent library

The first method use Dirent library, GetLocalDrives() method to be precise. The code in Listing 2 shows the use of the drive mask to find out what drives are connected to the computer. All results are saved to the file including optical drives etc.

```
DWORD uDriveMask = GetLogicalDrives();
while (uDriveMask){
   if (uDriveMask & 1)
      {
         diskFile << driver << endl;
      }
      uDriveMask >>= 1;
   }
}
```

Listing 2: Use of method GetLogicalDrives()

Using a system command

This technique makes use of the ability of C/C++ to call system commands. The method is thus much more concise:

```
void SearchingForDrives::b(){
   system("wmic logicaldisk get name > disk.txt");
}
```

Listing 3: Usage of system command

3.4.6 SearchingForFiles layer

Inputs	file disk.txt
Outputs	file list.txt

The purpose of this layer is to go through the found drives and look for files that are marked to be attacked. It opens the file containing the found drives (disk.txt) and searches each specified disk. Before it starts to locate the files, however, it determines whether the corresponding drive is fixed or removable. In case it is neither fixed nor removable, the search of that drive is aborted. The disk checking code can be seen in Listing 4.

```
boolean toCheck = false;
LPCSTR drive = path;
switch (GetDriveType(drive))
{
   case DRIVE_FIXED:
   {
      toCheck = true;
      break;
   }
   case DRIVE_REMOVABLE:
   {
      toCheck = true;
      break;
   }
   case DRIVE_NO_ROOT_DIR:
   {
      break;
   }
   ...
}
```

Listing 4: Disk checking

If a drive is recognized as fixed or removable, the algorithm proceeds with the search. The found files are stored in file 'list.txt'.

An example of the file content is displayed in Fig. 12.

The class also accommodates two methods that can find the files in two distinct ways.

Using a system command One method takes advantage of the system-command call again. You can see the command in Listing 5. The variable 'path' contains a single drive label, i.e. 'C:'.

Usage of WIN32_FIND_DATA with delayed writing

Figure 12: Example of of list.txt

```
427 D:\bitbucket\staz\java\dev-project\alfresco_4_1_4\alfresco\WEB-INF\classes\alfresco\desktop\Alfresco.exe
428 D:\bitbucket\staz\java\dev-project\alfresco_4_1_4\WebContent\WEB-INF\classes\alfresco\desktop\Alfresco.exe
429 D:\bitbucket\staz\java\install-packages\alfresco_sdk_4_1_4\lib\server\build\alfresco\desktop\Alfresco.exe
430 D:\bitbucket\staz\java\install-packages\alfresco_sdk_4_1_4\lib\server\config\alfresco\desktop\Alfresco.exe
431 D:\bitbucket\staz\java\install-packages\alfresco_sdk_4_1_4\lib\server\source\alfresco\desktop\Alfresco.exe
432 D:\bitbucket\staz\java\install-packages\alfresco_sdk_4_1_5\lib\server\config\alfresco\desktop\Alfresco.exe
433 D:\bitbucket\staz\java\install-packages\alfresco_sdk_4_1_5\lib\server\source\alfresco\desktop\Alfresco.exe
434 D:\bitbucket\staz\java\dev-project\alfresco_4_1_4\build\temp\WEB-INF\classes\alfresco\desktop\Alfresco.exe
435 F:\glview428.exe
436 F:\postgresql-9.3.5-3-windows-x64.exe
437 F:\tcm851ax32_64.exe
438 F:\drivers\1hvh14w8.exe
439 F:\drivers\n1avu35w.exe
440 F:\drivers\nvga202w8164.exe
441 F:\Portal\Portal.exe
442 F:\Portal\Portal_cz.exe
443 F:\drivers\Carmera\Setup.exe
444 F:\drivers\Energy_Management\setup.exe
445 F:\drivers\OneKeyTheater\setup.exe
446 F:\drivers\OneKey_Recovery\setup.exe
447 F:\drivers\Touchpad\Setup.exe
448 F:\drivers\Wlan\Autorun.exe
```

```
"dir " + path + "\\*.exe /s /b >> list.txt";
```

Listing 5: The system command that is used to seek files

This way of searching files is programatically more difficult than using the system command. First we need to find all directories that could possibly contain the sought files. An example can again be found below in Listing 6.

The specified directories are stored in a list serving as FIFO, i.e. a queue. The directories in the queue are searched one by one while adding all subdirectories to the end of the queue. So each directory is first checked if it contains any subdirectories and only then if it contains files. All found EXE files are added to a list (a different one). When all directories are searched, we obtain the complete list of found files, which is then written to a file.

```
WIN32_FIND_DATA FindFileData;
HANDLE hFind = FindFirstFile(path, &FindFileData);
list<wstring> output;
if (hFind != INVALID_HANDLE_VALUE)
{
   do
   {
      if (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){ ... }
   }
}
```

Listing 6: Finding new directories using WIN32_FIND_DATA

This way of scanning does not cover all kinds of folders. There are special folders that are

unnecessary (and may even lead to infinite looping) to search so a check is in place to exclude them.

To give an overview. Consider directory 'D:\myDir'. If I was searched by the code from Listing 6, the subdirectories would be: 'D:\myDir\.', 'D:\myDir\..' and 'D:\myDir\$RECYCLE.BIN'. It makes no sense to include those directories in the search. Files in 'RECYCLE.BIN' will most likely be deleted so it is ineffective to invest valuable execution time infecting them. Directory '.' denotes the current directory that is being searched. Folder '..' marks the parent directory. Thanks to utilizing queue in the algorithm we can be sure it had been already searched.

Our goal is to also find the files. The process is almost identical to finding directories as you can se in Listin 7. Searching for files is limited to only those specified by the user. These have to be EXE files.

```
WIN32_FIND_DATA FindFileData;
HANDLE hFind = FindFirstFile(path, &FindFileData);
list<wstring> output;
if (hFind != INVALID_HANDLE_VALUE)
{
   do
   {
      if (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){ }
      else {
         ...
         wstring suffix = file.substr(file.size() - 3, file.size());
         if (suffix == L"exe"){
            output.push_back(file);
         }
      }
   }
}
```

Listing 7: Searching EXE files using WIN32_FIND_DATA

Once the folder seach is over and all tester files are found the list of these is written to the output file 'list.txt'.

Using WIN32_FIND_DATA with immediate writing

The logic of this approach is the same as the above, the difference being that the found files get output immediatelly when discovered.

3.4.7 FileToInfectHandler layer

Inputs	file list.txt
Outputs	—

This layer's only purpose is switching between serial and parallel way of infection.

Serial approach The files are infected one at a time – each time a file path from 'list.txt' is loaded and the matching file gets infected.

Parallel approach The file paths are loaded at once this time, but the infection is executed in separate threads. See Listing 8.

```
while (getline(files, path)){
   thread t(callFunction, index, path);
   t.join();
}
```

Listing 8: Parallel infection

3.4.8 InfectionChecker layer

InfectionChecker verifies that files marked for infection are not yet infected. It cosists of three methods. Each one of them checks a different infection indicator, but they all return true if the given file is already infected, false otherwise.

Prepended: This method checks whether the given file is bigger in length than the virus file. If not false is returned. Otherwise, the proximity of the virus code's end is examined. If there is any content past it the method returns true. It is designed for prepending viruses.

Length check – equal: The method checks the length of the passed file. In case it has the same length as the infected file, true is returned, false otherwise.

Length check – greater: Much like the length checking method this one also checks the length of the file though with a slight adjustment. If the length is equal or greater than the infected file, true is returned, false otherwise.

3.4.9 InfectionExe layer

Inputs	file path to a tester file
Outputs	infected file

InfectionExe contains various methods designed to infect files. The indexes used to access methods of this layer are the same as the ones used in SeparationExe because the separation depends on the type of infection. The layer should be capable of infecting all of the files in 'list.txt'. Althought there are many ways in which a file can be infected, this layer contains just three of them.

- **Prepending**

  Prepending method is based on the principle of prepending the virus to the front of the original code. Detailed description can be found in chapter 2.1.2 on page 15.

  First, the targeted file is opened in read mode and loaded to memory as a string. The same goes for the body of the virus. Then the targeted file is opened again only this time in write mode. The virus is then written to the front and finally the original file is saved and closed.

- **Overwriting**

  This type of infection is slightly simpler. The given file is opened for writing and its content is overwritten by the virus code. The original source code is lost and there is no way of recovery.

- **Parasitical**

  This is similar to the prepending method, but it does not move the file content behind the virus. It replaces the front part of the original code with the the virus body, the extracted part being inserted to the very end of the file. As soon as the infected file is executed, the virus section removes itself from the front and puts the original code back together. Detailed description is to be found in chapter 2.1.2 on page 16.

3.4.10 Payload layer

The payloads of the viruses are significantly simplified. They should not cause any harm or compromise data. Payload gets executed near the end of virus life cycle. The reason is described in chapter 2.1.2 on page 14. Though these methods do not carry out any useful activity, they do vary in the number of characters they are comprised of, the execution time – here comes evolution into play to select the shortest and the fastest one. There are six possible payload

types as laid out below.

- **Message method:** The message method writes a simple sentence to the console: 'I was here. Yours sincerely, Virus' .
- **Do-nothing method:** The do-nothing method contains no code and thus does nothing.
- **Inner-cyles method:** This method contains two nested cycles looping from 1 to 100. This is intended to take more time to execute than the two previously described methods so evolution should not select the offspring containing this method as the best one.
- **Alphabet method:** The alphabet method uses a cycle to generate an alphabet of the form: 'AaBbCc...' and prints it in to the console.
- **File-writing method:** This method opens (or creates) a file and writes a line of text to it. Afterwards it opens the file, reads the line that was just input and prints it to the console.
- **Fibonacci method:** This method calculates a portion of Fibonacci sequence and prints it to the console.

3.4.11 SS life cycle

The life cycle of a SS is the time from execution start to its end. The layers do their work one by one in the sequence defined in Virus class. The workflow is depicted in Fig. 13.

The 'Layer id' attribute controls the order in which the layers are executed. The sequence identical to the one defined in the body of the correspondinf SS – having the form of a string consisting of layer ids and colons separating them.

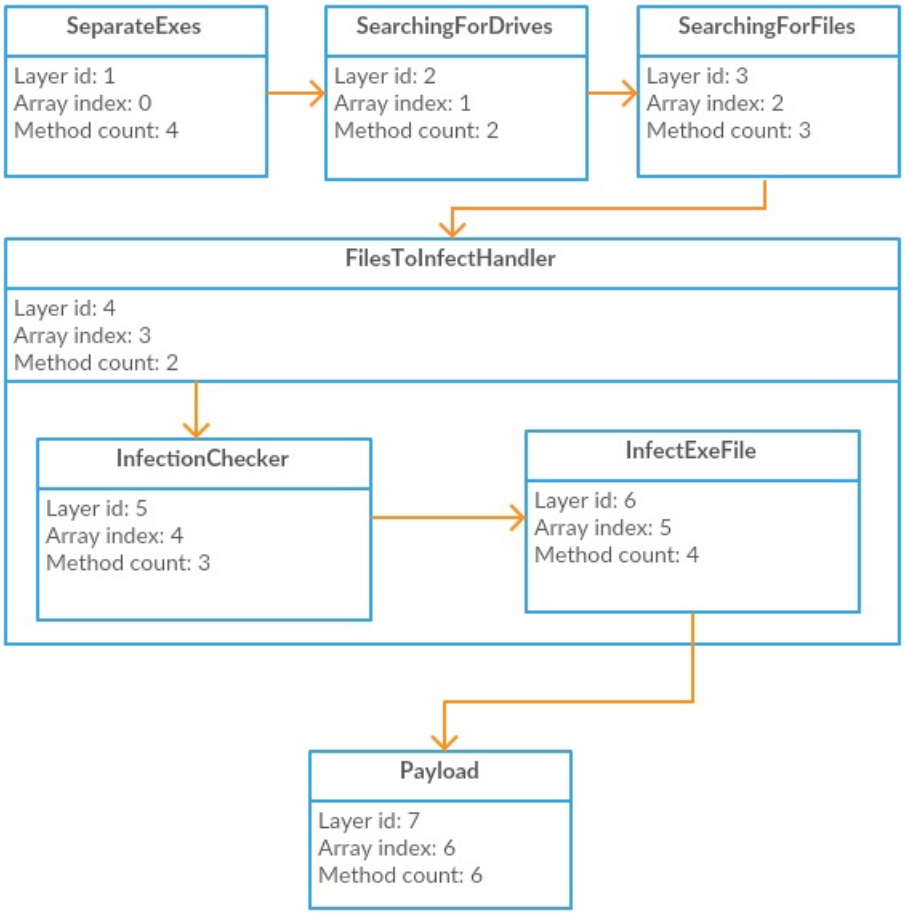
The 'Array id' attribute is the index in the array used by the program.

The last attribute is 'Method count', which determines the count of methods in the given layer. Only one of those methods may be used per layer (except for Payload layer – described in section 3.4.1).

3.5 Overall functionality of the program

The flow of the program as is may be difficult to understand. Its simplified version is therefore presented in Fig. 14 on page 43. Some parts are essential, however (e.g. fitness calculation). The key functionality is fully explained in subsections of this chapter.

Figure 13: Life cycle of a SS



Before execution the constants have to be initialized with:

- The file to store summary records (summary.txt).
- The execution path of the program.
- The size of the executing file.
- The initalized Virus class.
- The maximum number of methods for each layer.
- The control values from SOMAconfig.csv (described in detail in subsection 3.2).
- The files to be spread across the device (described in detail in subsection 3.3.2 on page 28).
- The calculated execution time of the file-searching methods (described in detail in subsection 3.3.2 on page 28

- The generated random coordinates for each offspring.
- The computed fitness values – one for each offspring.
- The first leader.

After the constants are set, the tester files are spread across the specified drives (see 3.3.2). Then the constants created to hold file-searching methods' execution time are set (see 3.3.2).

At this moment, all necessary constants are set and offsprings can be generated.

The first generation si generated randomly.

3.5.1 Generation offsprings

To successfully generate offsprings there are some supporting structures. One of them is maxMethodCount, an array that holds the number of methods used in each layer. For example, 'maxMethodCount[0] = x' tells us that x-th method is used for the first layer. That means that the offspring can have the method index ranging from 0 to method count - 1 (e.g. a layer comprising 2 methods would have those methods indexed with 0 and 1). The x is the index of the method used in the given layer.

The index of the used method is generated as shown in Listing 9 on page 44.

In some layers, the indexes have special behaviour. The method in sixth layer (the layer with id 5) has to have the same index as the method in the first layer. The sixth layer handles file infecting, while the first one handles recovery of the infected file. Should the separation layer be successful in recovering the infected files, the indexes, and so the methods as well, have to correspond to those used in the infection layer.

The seventh layer is also special because it handles payloads. The difference between the payload layer and other layers is that more than one method can be used (see 3.4.1). The index of the method is generated first and then the method itself follows (there may be more than one index/method). If more than one payload is used, it may so happen that the selected methods will be the same – it depends solely on evolution. There is also the possibility that the layer will have no payload methods at all.

3.5.2 Fitness calculation

After setting offspring methods fitness can be computed. The whole process is managed by FitnessHandler class as described in chapter 3.3.1 on page 26 page.

3.5.3 Leader selection

Once all offsprings are evaluated the algorithm may proceed with the leader selection. The leader is an entity with the lowest cost function value (the highest fitness). Such an entity is marked as the leader of the generation and does not move in the next migration loop, nonetheless, it becomes still a member of the new generation.

3.5.4 New position selection

When the leader is found, all other offpsrings start searching their new position. Once they find one, the fitness of this position is calculated. If the fitness is better, the position is stored. After all positions are considered, the best one is taken and the offspring is migrated to it – new coordinates and a new fitness value are assigned to that offspring.

The code in Listing 10 on page 45 illustrates the computation that takes place in order to find a new position.

The prtVector is generated for each offspring simulating mutation, which is typical for biological viruses. The value tmpCoords are the new temporary coordinates that have to be tested. As soon as the fitness of tmpCoords is calculated, it is compared to the original offspring's fitness. If the new fitness value is better, the coordinates are stored and the next position is tested. At the end, when all of the positions have been tested, the algorithm checks whether any better positions were found. In case no better position could be found the last calculated position is taken. if the new fitness value is the same as the one of the original entity, it is as if it was worse.

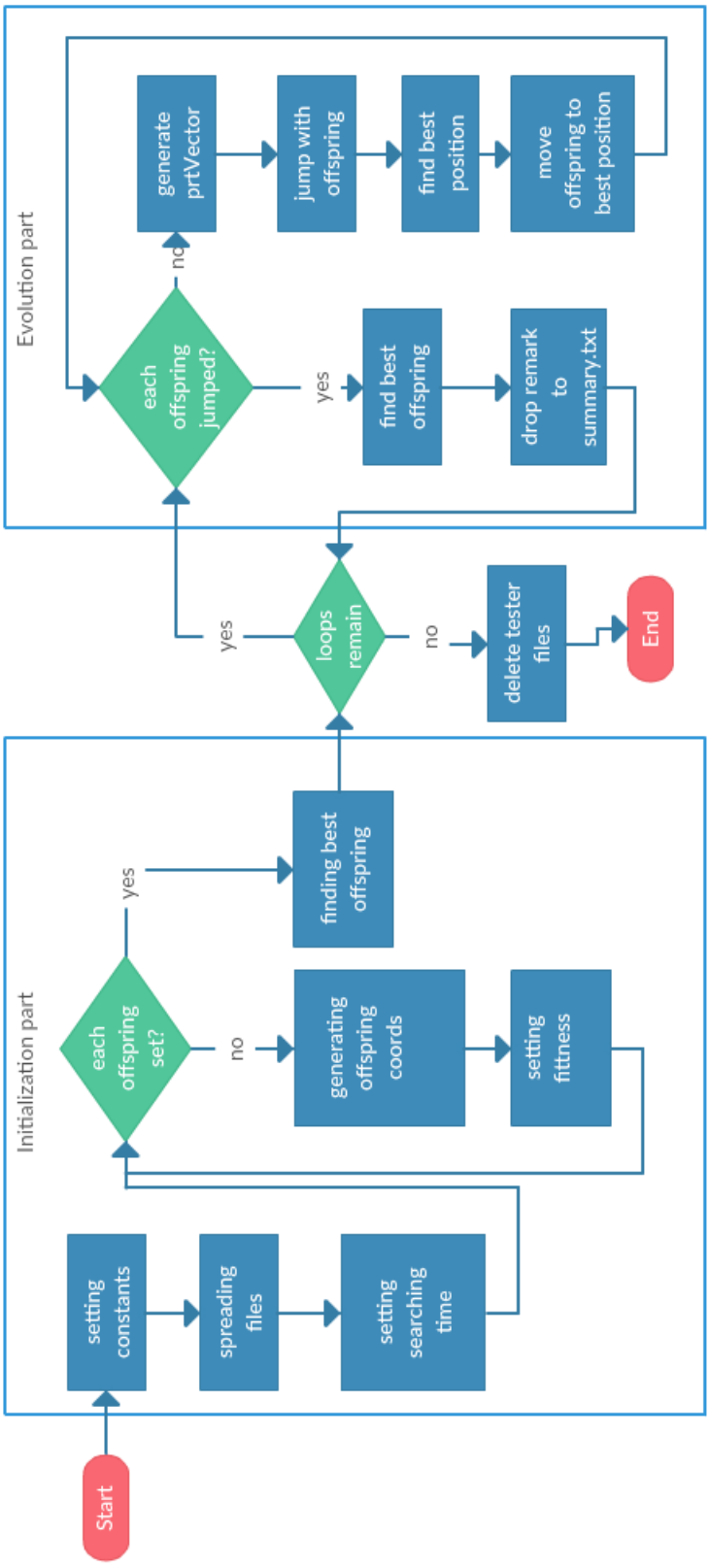
After new positions of all offsprings are found, a leader is selected and the whole process is repeated.

It is possible that the leader of several consecutive generations remains the same, but it does not mean that the desired extreme was found. It may take some time for other offsprings to migrate and find a better way.

3.5.5 End of evolution

The program terminates when the set number of migration loops is reached. The best solution is the leader of the last generation. All results are saved in the summary file which is opened immediately after that.

Figure 14: Program flow



```
for (int j = 1; j < layerCount; j++){
  if (j == 5){//indexes of separating and infecting layers have to be same
    int coord = abs(rand() % maxMethodCount[j]);
    offsprings[i].coords[j] = coord;
    offsprings[i].coords[0] = coord;
  }
  else if (j == 6){
    int count = abs(rand() % ConstantsHandler::getPayloadCount());
    offsprings[i].payloadMethodsUsed = count;
    for (int m = 0; m < count; m++){
      offsprings[i].coords[j + m] = abs(rand() % maxMethodCount[6]);
    }
  }
  else {
    offsprings[i].coords[j] = abs(rand() % maxMethodCount[j]);
  }
}
```

Listing 9: Generating an offspring

```
setting prtVector();
for (int i = 1; i < layerCount; i++) {
  if (i == 5){
    int newCoord = myOf.coords[i] + abs((best.coords[i] - myOf.coords[i]) *
      * step * t) *prtVector[i];
    newCoord = newCoord % maxMethodCount[i];
    tmpCoords[i] = newCoord;
    tmpCoords[0] = newCoord;
  }
  else if (i == 6){
    payloadUsed = myOf.payloadMethodsUsed + abs((best.payloadMethodsUsed -
      - myOf.payloadMethodsUsed) * step * t * prtVector[i];
    payloadUsed = payloadUsed % (ConstantsHandler::getPayloadCount());
    for (int m = 0; m < payloadUsed; m++){ //for each payload
      int newCoord;
      if (myOf.payloadMethodsUsed <= m){
        newCoord = abs(rand() % maxMethodCount[i]);
      }
      else{
        newCoord = myOf.coords[i + m] + abs((best.coords[i + m] - myOf.coords[i
            + m]) * step * t) *prtVector[i + m];
        newCoord = newCoord % maxMethodCount[i];
      }
      tmpCoords[i+m] = abs(newCoord);
    }
    break; //payload is the last layer - all have been set
  }
  else {
    int newCoord = myOf.coords[i] + abs((best.coords[i] - myOf.coords[i]) *
       step * t) *prtVector[i];
    newCoord = newCoord % maxMethodCount[i];
    tmpCoords[i] = newCoord;
  }
}
```

Listing 10: Determining a new position

4 Test results

Several tests were performed. Configuration attributes of each test are presented and the obtained results are commented.

4.1 Test no. 1

Setting:

Table 6: Test no. 1, attributes

Offspring count	15
Number of tester file copies	5
Drives to infect	D
Step	0.36
PRT value	0.5
PathLength	4
Number of migration loops	15

4.1.1 Shorter file

The file that was used for this test was shorter than the virus file.

The leader of the first generation was an offspring with the cost function value equal to 30107. The body of the leader is described below:

SeparateExe layer: 2 (the method for rewriting virus)
SearchingForDrives layer: 1 (the method using GetLogicalDrives())
SearchingForFiles layer: 2 (the method using recursive approach with HANDLE, file paths are written to the resulting file at the end – when all of them are found)
FilesToInfectHandler layer: 1 (the method using serial approach to infecting files)
InfectionChecker layer: 2 (the method checking length – equal)
InfectExe layer: 2 (the virus rewriting method)
Payload layer: 5 (the message method)

The last leader's cost function value was 29854. Its body is described below:

SeparateExe layer: 2 (the method for rewriting virus)
SearchingForDrives layer: 1 (the method using GetLogicalDrives())
SearchingForFiles layer: 2 (the method using recursive approach with HANDLE, file paths are written to the resulting file at the end – when all of them are found)
FilesToInfectHandler layer: 2 (the method using parallel approach to infecting files)
InfectionChecker layer: 2 (the method checking length – equal)
InfectExe layer: 2 (the virus rewriting method)

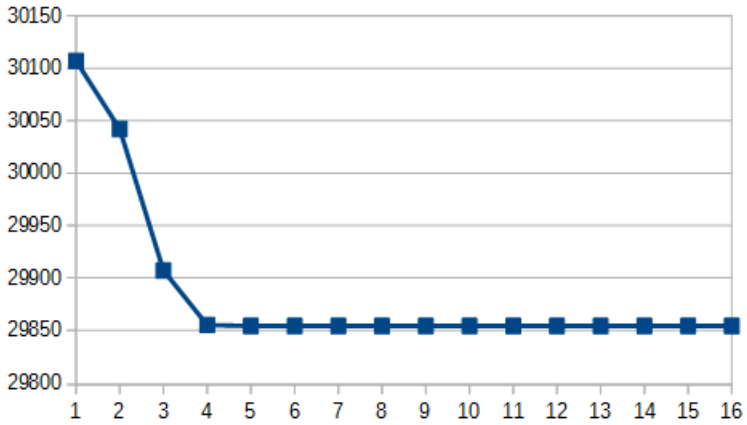
Comparison

As we can see there is progress – the cost function was lowered by 253. The first leader had a payload method. which is clearly not the best SS. The last leader from the test ended up having no payload methods. As early as in the first generation the rewriting method for infection was used – this is the fastest method from this layer. As we can see the last leader uses parallel approach to infecting files. Because the rewriting method was used, the selected infection checker method checks the length only. This is a fast method and is suitable for this infection type.

Cost function progression

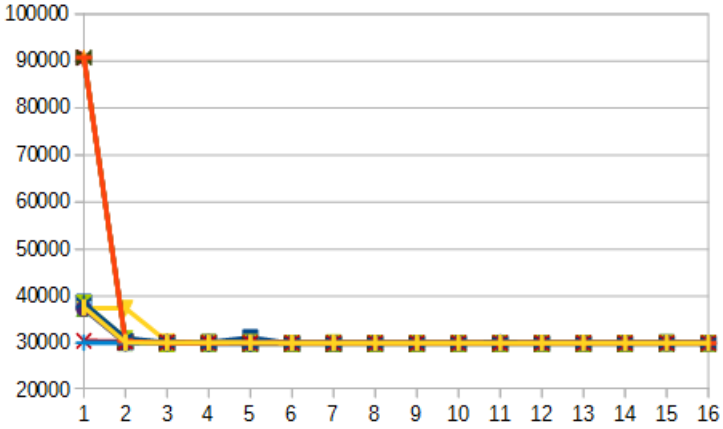
Fig. 15 depicts the progession of the cost function.

Figure 15: The cost function of leaders – test no. 1, shorter tester file



As the figure shows the best cost function was found in the fourth iteration. Fig. 16 displays the migration of each entity during evolution. We can see that during the first three migrations all entities converged nearly to the best possible solution.

Figure 16: The migration of entities – test no. 1, shorter tester file



4.1.2 Longer file

For this test tester files longer than virus files were chosen. Because one of the cost function's attributes is execution time, it is expected that the cost function will return higher values than we saw in the shorter file test. The resulting leader's body should be the same or very similar though.

In the first generation the leader was an offspring with the cost function value equal to 37712. The body of the leader is described below:

SeparateExe layer: 2 (the method for rewriting virus)
SearchingForDrives layer: 2 (the method using the system command)
SearchingForFiles layer: 3 (the method using recursive approach with HANDLE, files paths are written to the resulting file immediately – when they are discovered)
FilesToInfectHandler layer: 1 (the method using serial approach to infecting files)
InfectionChecker layer: 1 (the checking method suitable for prepending viruses)
InfectExe layer: 2 (the virus rewriting method)

The last leader has the cost function value equal to 37566. Its body is described below:

SeparateExe layer: 2 (the method for rewriting virus)
SearchingForDrives layer: 1 (the method using GetLogicalDrives())
SearchingForFiles layer: 3 (the method using recursive approach with HANDLE, files paths are written to the resulting file immediately – when they are discovered)
FilesToInfectHandler layer: 1 (the method using serial approach to infecting files)
InfectionChecker layer: 2 (the method checking length – equal)
InfectExe layer: 2 (the virus rewriting method)

Comparison

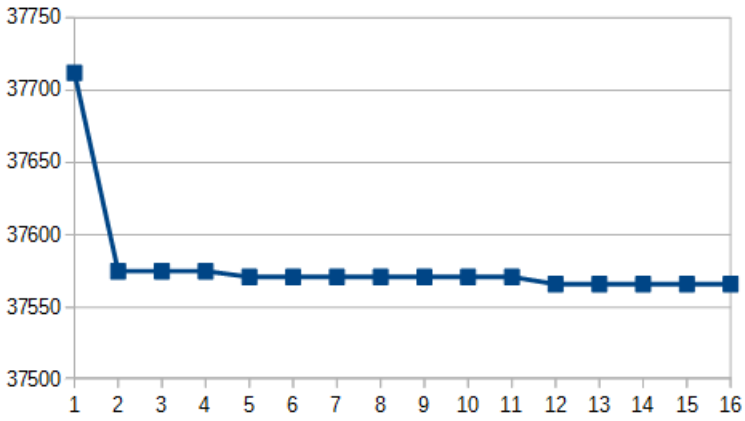
Both leaders have similar bodies and neither one contains payload methods. They both use the rewriting method to infect files. The difference is in the drive-searching methods where the system command appears to be slower. Furthermore they differ in InfectionChecker layer, here the method performing fewer comparisons was selected for the last leader.

Cost function progression

Fig. 17 depicts the progession of the cost function.

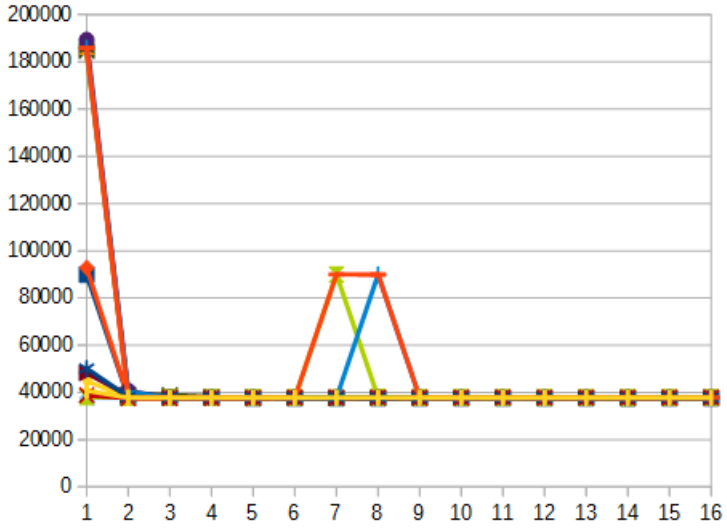
The cost function was lowered by 146. We can observe a different proggresion than in the case of shorter files – significant change occurred in the first iteration, but the final cost function was not found until 12th iteration. As expected the cost function values were higher, but the bodies were much the same – both SSs do not have any payload methods and both use the

Figure 17: The cost function of leaders – test no. 1, long tester file



rewriting method of infection. The progression of the cost function of all entities is shown in Fig. 18. We can see that some entities jump far from the leaders between 6th and 9th migration loop. This was caused by the entities getting stuck – the new entity's position can't be the same as the leader's so a new position is picked even if it is worse. This behaviour was programmed to avoid conversion because the results are integers and it is likely that the entities will converge to one position. the

Figure 18: The migration of all entities - test no. 1, long tester file



4.2 Test no. 2

Setting:

Table 7: Test no. 2, attributes

Offspring count	7
Number of tester file copies	5
Drives to infect	D
Step	0.36
PRT value	0.5
PathLength	4
Number of migration loops	15

4.2.1 Shorter file

The file that was used for this test was shorter than the virus file.

The leader of the first generation was an offspring with the cost function value equal to 17697. The body of this leader is described below:

SeparateExe layer: 1 (the method for prepending virus)
SearchingForDrives layer: 1 (the method using GetLogicalDrives())
SearchingForFiles layer: 1 (the method using the system command)
FilesToInfectHandler layer: 2 (the method using parallel approach to infecting files)
InfectionChecker layer: 3 (the method checking length – greater)
InfectExe layer: 1 (the method for prepending virus)

The last leader's cost function value was 17375. Its body is described below:

SeparateExe layer: 2 (the method for rewriting virus)
SearchingForDrives layer: 1 (the method using GetLogicalDrives())
SearchingForFiles layer: 1 (the method using the system command)
FilesToInfectHandler layer: 2 (the method using parallel approach to infecting files)
InfectionChecker layer: 1 (the checking method suitable for prepending viruses)
InfectExe layer: 2 (the virus rewriting method)

Comparison

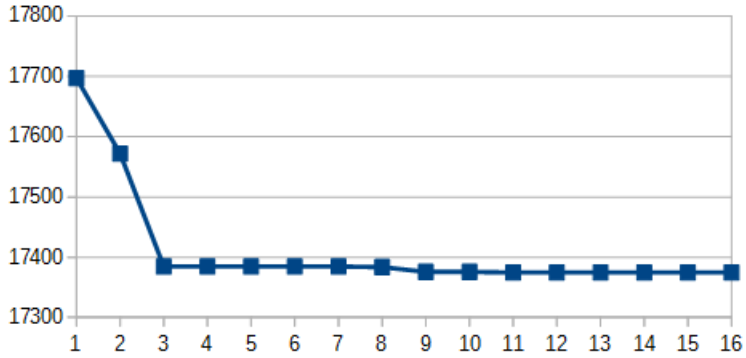
The cost function was lowered by 322. The first leader's body contained the method for prepending SS, which is less time-efficient than the method for rewriting SS. The latter is used in the body of the last leader. InfectionChecker layer of the first leader utilizes the method that is suitable for prepending SS. The last leader, on the other hand, uses the rewriting method, which is

more efficient.

Cost function progression

Fig. 19 depicts the progression of the cost function.

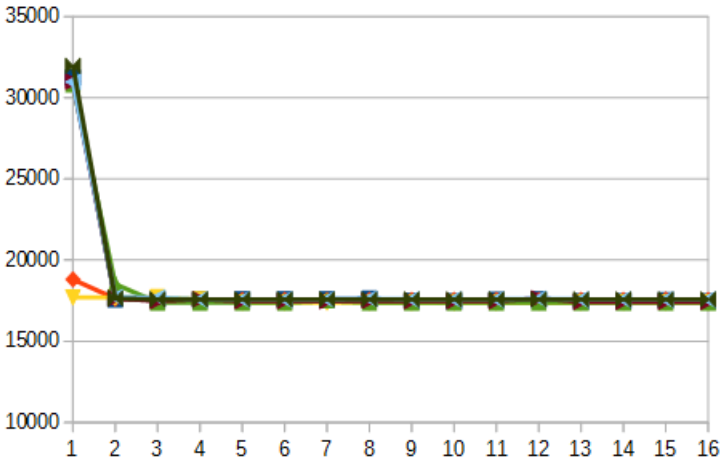
Figure 19: The progression of the cost function – test no. 2, shorter tester file



As the figure shows the best cost function value was found in 11th iteration although the algorithm was very close to discovering the best value after the third one.

The migration process of all entities illustrates Fig. 20. It can be seen that after just three migrations all of the entities were close to the best value.

Figure 20: Migration of all entities – test no. 2, shorter tester file



4.2.2 Longer file

For this test tester files longer than virus files were chosen.

In the first generation the leader was an offspring with the cost function value equal to 18569. The body of the leader is described below:

SeparateExe layer: 2 (the method for rewriting virus)
SearchingForDrives layer: 1 (the method using GetLogicalDrives())
SearchingForFiles layer: 1 (the method using the system command)
FilesToInfectHandler layer: 2 (the method using parallel approach to infecting files)
InfectionChecker layer: 3 (the method checking length – greater)
InfectExe layer: 2 (the virus rewriting method)
Payload layer: 6 (the method computing Fibonacci's sequence, 100 iterations)
Payload layer: 3 (the method going through two nested loops, 100 steps each)
Payload layer: 4 (the method writing the alphabeth)

The last leader has the cost function value equal to 18096. Its body is described below:

SeparateExe layer: 2 (the method for rewriting virus)
SearchingForDrives layer: 1 (the method using GetLogicalDrives())
SearchingForFiles layer: 1 (the method using the system command)
FilesToInfectHandler layer: 2 (the method using parallel approach to infecting files)
InfectionChecker layer: 2 (the method checking length – equal)
InfectExe layer: 2 (the virus rewriting method)

Comparison

The cost function was lowered by 473. Both leaders use rewriting methods for infecting files. The first leader had some payload methods which slowed down its execution. The last leader does not have any payload methods.

Cost function progression

Fig. 21 depicts the progression of the cost function.

As the figure shows the best cost function value was found in 11th iteration, nevertheless after 9 iterations the leader's value was very close to the best one.

The migration process of all entities illustrates Fig. 22. Although the entities were far from the best value during the first four migrations, the situation drammatically changed after the fourth migration and the entities were spread out near the best value.

Figure 21: The course of the cost function – test no. 2, longer tester file

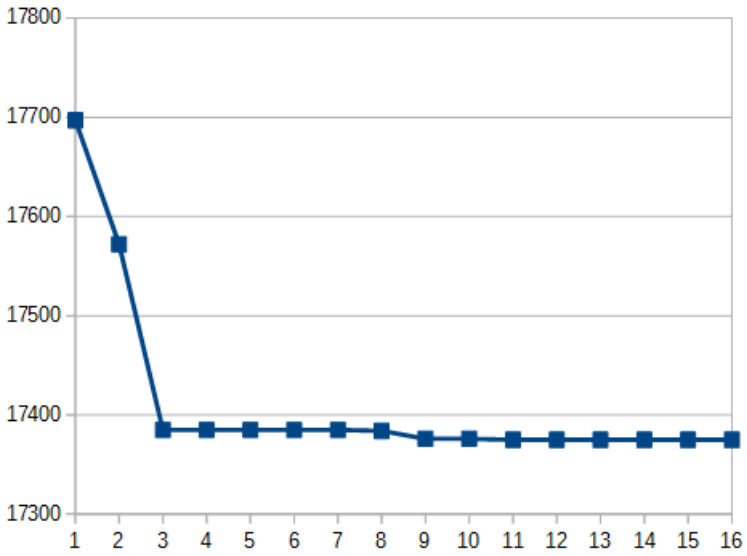
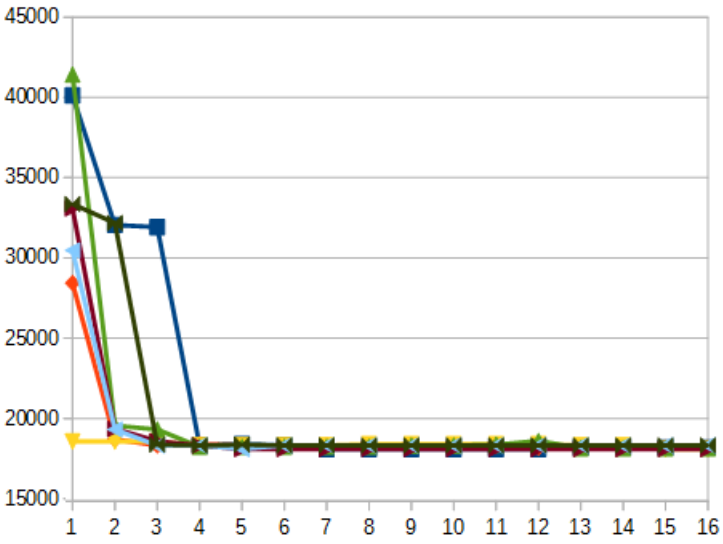


Figure 22: Migration of all entities – test no. 2, longer tester file



4.3 Summary of methods in bodies of entities

Following Table 8 shows the count of each method in each layer that were used in tests desribed above. The data represent all entities from all migration loops. These methods were summed up to get the final result. The rows with the most frequent method are highlighted for each layer.

Fig. 23 is there to illustrate this layout. X-axis represents layer names and each coloured column means one method in this layer. Y-axis tells us the method count.

As Fig. 23 shows the frequency of method usage is varied. The difference is most distinct

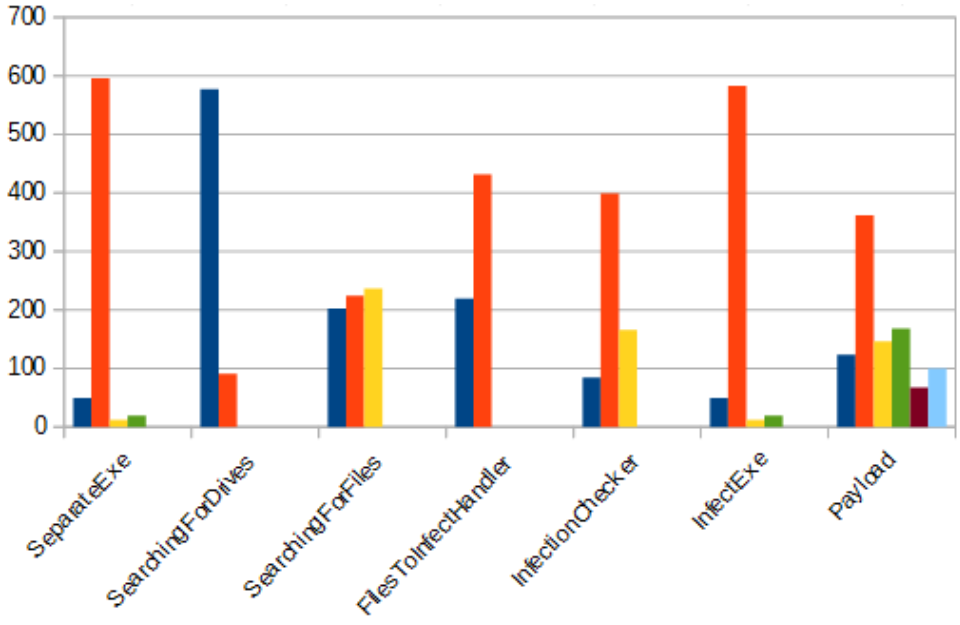
Table 8: Description of used methods

Layer	Method	Count	Percentage in layer
SeparateExes	prepended	49	7.25
	rewriting	596	88.17
	parasitical	12	1.78
	fake	19	2.81
SearchingForDrives	GetLogicalDrives()	577	86.38
	system command	91	13.62
SearchingForFiles	system command	202	30.51
	FindNextFile() - delayed entry to file	224	33.83
	FindNextFile() - immediate entry to file	236	35.65
FilesToInfectHandler	serial approach	219	33.64
	parallel approach	432	66.36
InfectionChecker	prepending file	84	12.94
	length check - equals	399	61.45
	length check - larger	166	25.58
InfectExe	prepended	49	7.25
	rewriting	596	88.17
	parasitical	12	1.78
	fake	19	2.81
Payload	message	123	12.73
	nothing-doing method	362	37.47
	inner-cycles method	146	15.11
	alphabeth	168	17.39
	writing to file	67	6.94
	fibonaci method	100	10.35

in InfectExe layer. The rewriting method is made of fewer characters and has the shortest execution time. That is why it is the most commonly used method.

On the other hand, methods in Payload layer are used with almost equal frequency. Most commonly used method is the do-nothing method. This method is empty therefore it does not contain any characters and needs no execution time. If we compare it to the method writing a sentence to a file, we can clearly see that it consumes much more time.

Figure 23: Methods used by entities



4.4 Summary of methods in the leaders' bodies

Following Table 9 shows the count of every method in all the layers that were used in leaders' bodies. The data include all leaders from all migration loops and all tests. These methods were summed up to get the final result. The rows with the most frequent methods in each layer are highlighted.

Fig. 24 is there to illustrate this layout. X-axis represents layer names and each coloured column means one method in this layer. Y-axis tells us the method count.

As Fig. 24 shows the differences among method frequencies are more apparent because the leaders are selected from entities and the methods used in their bodies are swapped for the better ones as evolution progresses.

In InfectExe some methods were not selected at all to be a part of a leader.

On the other hand, the method frequencies from Payload layer are quite similar.

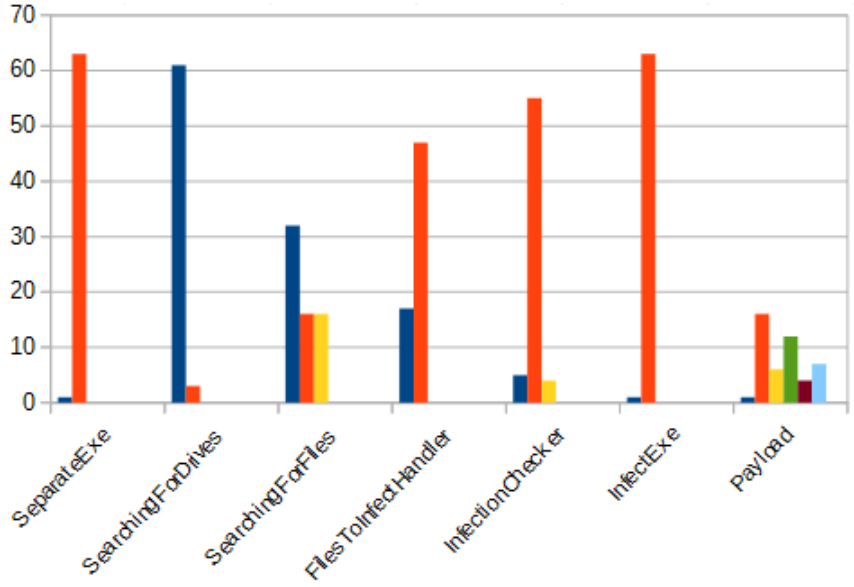
4.5 Payload layer methods frequencies

During all migration loops across all tests the number of entities was 664, the number of used payload methods was 966. As to leaders – 64 leaders were generated and only 46 payload methods were used in total.

Table 9: Description of methods used in leaders' bodies

Layer	Method	Count	Percentage in layer
SeparateExes	prepended	1	1.56
	rewriting	63	98.43
	parasitical	0	0
	fake	19	2.81
SearchingForDrives	GetLogicalDrives()	61	95.31
	system command	3	4.69
SearchingForFiles	system command	32	50
	FindNextFile() - delayed entry to file	16	25
	FindNExtFile() - immediate entry to file	16	25
FilesToInfectHandler	serial approach	17	26.56
	parallel approach	47	73.44
InfectionChecker	prepending file	5	7.81
	length check - equals	55	85.94
	length check - larger	4	6.25
InfectExe	prepended	1	1.56
	rewriting	63	98.44
	parasitical	0	0
	fake	0	0
Payload	message	1	2.17
	nothing-doing method	16	34.78
	inner-cycles method	6	13.04
	alphabeth	12	26.09
	writing to file	4	8.7
	fibonaci method	7	15.22

Figure 24: Methods used by leaders



5 Conclusion

At the beginning of the thesis SS was defined and SOMA algorithm was chosen for evolution. SS was divided into logical parts – layers that cooperate. Then specific methods for each layer were defined. The definition of parameters was separated into an external file in order to make the usage comfortable for the user. The output format was created in a way that is suitable and comprehensible for the user too.

At the end, when the whole program was thoroughly described, tests were carried out. The results of these were summarized and depicted using charts and tables.

Results obtained from the tests weren't very surprising. Leaders' bodies were comprised of methods that were the fastest and the shortest possible as defined in the cost function definition section.

I found the work on the thesis very interesting. I became familiar with the topic of viruses and computer malware. I wrote several simple viruses that attacked EXE files. After that I used their code, split it and put to the relevant layers. The biggest challenge was to handle the cost function rating.

The thesis was a valuable lesson to me and I am very glad that I chose this topic. I am very glad that I could work with prof. Ing. Ivan Zelinka, Ph. D. and I am looking forward to working on my dissertation under his supervision.

References

[1] Bc. Igor Hák: Moderní počítačové viry, third edition. 2005

[2] *NORTON^{TM} - Antivirus Software and Spyware Removal*, [online]. [cit. 5.4.2017]. Available from: https://us.norton.com/internetsecurity-malware-what-is-a-computer-virus.html

[3] *Next-Gen Cybersecurity Threat Intelligence / Webroot*, [online]. [cit. 5.4.2017]. Available from: https://www.webroot.com/us/en/home/resources/articles/pc-security/computer-security-threats-computer-viruses

[4] FILIOL, Eric. *Computer viruses: from theory to applications*. New York: Springer, 2005. ISBN 9782287239397.

[5] *Symantec - Global Leader In Next-Generation Cyber Security*, [online]. [cit 25. 4. 2017] Available from: https://www.symantec.com/security_response/publications/monthlythreatreport.jsp

[6] *Antivirus for Windows, Mac and Android - Panda Security*, [online]. [cit 14. 4. 2017] Available from: http://www.pandasecurity.com/homeusers/security-info/classic-malware/virus/

[7] GODFREY C. ONWUBOLU and B.V. BABU. *New optimization techniques in engineering*. Berlin: Springer, 2004. ISBN 9783642057670.

[8] *Evolution Definition in the Cambridge English Dictionary*, [online]. [cit. 15. 4. 2017] Available from: http://dictionary.cambridge.org/us/dictionary/english/evolution

[9] ZELINKA, Ivan. *Evoluční výpočetní techniky: principy a aplikace*. Praha: BEN - technická literatura, 2009. ISBN 978-80-7300-218-3.